From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures

Miguel Correia, Nuno Ferreira Neves,
Paulo Veríssimo

DI–FCUL TR–04–5

June 2004

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures *

Miguel Correia **Nuno Ferreira Neves** **Paulo Veríssimo**
Faculdade de Ciências da Universidade de Lisboa
Bloco C6, Piso 3, Campo Grande, 1749-016 Lisboa - Portugal
{mpc,nuno,pjv}@di.fc.ul.pt
Keywords: consensus, atomic broadcast, asynchronous systems,
Byzantine faults, intrusion tolerance, randomization

Abstract

This paper proposes a hierarchy of three Byzantine-resistant protocols aimed to be used in practical distributed systems: multi-valued consensus, vector consensus and atomic broadcast. These protocols are designed as successive transformations from one to another. The first protocol, multi-valued consensus, is implemented on top of a randomized binary consensus. The protocols share a set of important structural properties. Firstly, they do not use signatures obtained with public-key cryptography, a well-known performance bottleneck in this kind of protocols. Secondly, they are time-free, i.e., they make no synchrony assumptions, since these assumptions are often vulnerable to subtle but effective attacks. Thirdly, they have no leaders, thus avoiding the cost of detecting corrupt processes. Fourthly, they have optimal resilience, i.e., they tolerate $f = \lfloor \frac{n-1}{3} \rfloor$ out of a total of n processes. The multi-valued consensus protocol terminates in a constant expected number of rounds, while the vector consensus and atomic broadcast protocols have time complexities $O(f)$.

1 Introduction

Distributed protocols capable of tolerating Byzantine faults have been being studied for more than two decades [25, 21, 27, 2]. Recently, interest in these protocols has gained a new momentum under the designation of *intrusion tolerance* [33]. The basic idea is that the security concepts of attack, intrusion and vulnerability can be considered as *faults*, more precisely as arbitrary faults, also called Byzantine faults[1]. A consequence of this assertion is that Byzantine-resistant protocols can be important building blocks for the construction of secure systems.

*This work was partially supported by the FCT through project POSI/CHS/39815/2001 (COPE) and the Large-Scale Informatic Systems Laboratory (LASIGE).

[1]We follow the recent literature that uses interchangeably the terms 'Byzantine faults' and 'intrusions', or 'Byzantine-resistant' and 'intrusion-tolerant'. However, papers like [25, 21] consider accidental Byzantine faults, which are different from malicious Byzantine faults, i.e., intrusions. These latter faults should not be assumed to happen independently.

Byzantine-resistant (or intrusion-tolerant) protocols usually have higher time and message complexities than crash-tolerant protocols do. They are also more CPU-time demanding since they must use cryptography, and often public-key cryptography. This CPU-time issue is frequently dismissed since the processing power of computers is constantly increasing. However, new classes of computing environments are appearing in which resources are scarce, e.g., *embedded* and *ubiquitous computing*. This is an important motivation for the design of less CPU-time consuming intrusion-tolerant protocols. Moreover, public-key cryptography operations can be an important bottleneck for the performance of intrusion-tolerant systems even in more powerful hardware. Castro and Liskov designed an intrusion-tolerant NFS system which performs on average only 3% slower than standard NFS, in part due to avoiding the use of signatures based on public-key cryptography [9].

A central argument of this paper is that the design of efficient Byzantine-resistant protocols is crucial for the implementation of practical intrusion-tolerant systems, therefore these protocols have to avoid as much as possible the use of public-key cryptography. Moreover, practical intrusion-tolerant systems require protocols with other characteristics, like strict asynchrony, optimal resilience, and low time complexity. The paper provides a coherent family of protocols with these properties.

Paper Results. The paper presents a hierarchy of three Byzantine-resistant protocols: multi-valued consensus, vector consensus and atomic broadcast (see Figure 1). Consensus is a distributed systems problem with both theoretical and practical interest. The problem can be stated this way: how does a set of distributed processes achieve agreement on a value despite a number of process failures? The paper implements two flavors of consensus: *multi-valued consensus* that makes agreement on values with an arbitrary size; and *vector consensus* that makes agreement on a vector with the values proposed by several of the processes. An *atomic broadcast* protocol is a communication protocol that delivers the same messages to all processes and in the same order. Atomic broadcast is, for instance, the main component of fault-tolerant systems based on the state-machine approach, with both crash [29] and Byzantine faults [9]. Atomic broadcast has been shown to be equivalent to multi-valued consensus in systems prone to crash faults [19].

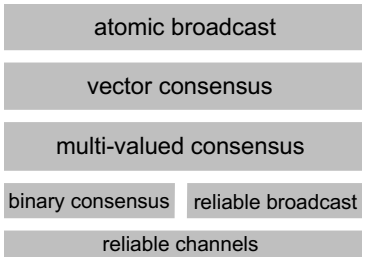

Figure 1: Protocol architecture.

The problem of consensus has been studied with different system models, such as the synchronous and the asynchronous time models, and with distinct types of failures, from crash to arbitrary. In asynchronous systems, consensus has been shown to be constrained by the FLP impossibility result, which says that it is impossible to solve consensus deterministically in a completely asynchronous system [17]. Consequently, various researchers have proposed ways to circumvent this limitation: using randomization [27, 2, 4, 30, 5, 8, 7], making synchrony

or timing assumptions on the behavior of the system [14, 16, 32], using failure detectors [10, 28, 13, 1, 12] or ordering oracles [26], using wormholes [11], or imposing conditions on inputs [24].

The protocols in the paper do not solve consensus from scratch but are built on top of a randomized binary consensus protocol (e.g., [4, 8]). They also require a reliable broadcast protocol (e.g., [5]). The three protocols share the following set of structural properties:

- *Signature free.* The protocols do not use signatures based on public-key cryptography.
- *Asynchrony.* The protocols are asynchronous, there are no synchrony assumptions whatsoever.
- *Distribution.* Decisions are taken in a distributed way, i.e., there are no coordinators, leaders or token-holders.
- *Optimal resilience.* The protocols tolerate $f = \lfloor \frac{n-1}{3} \rfloor$ faulty processes out of a total of n processes.

A hierarchy of protocols with this combination of characteristics is novel, to the best of our knowledge. Furthermore, these properties are provided coherently by all the protocols in the hierarchy. We argue that all of them are important if the protocols are to be used in practice. The argument for avoiding public-key cryptography (first property) has already been done above, so let us discuss the importance of the other three properties.

Many protocols in the literature are designated "asynchronous" but make synchrony assumptions, either explicitly [14, 16, 32] or contained in the unreliable failure detector abstraction [28, 13, 1, 12]. These assumptions can make the protocols vulnerable to subtle but effective attacks in the domain of time, something that cannot happen in time-free systems. Some discussion about this kind of attacks and the corresponding vulnerabilities can be found in [9, 7]. Our protocols are time-free or strictly asynchronous (second property) but circumvent FLP by being built on top of a randomized binary consensus protocol. Although randomized, the probability of termination of this protocol gets close to 1 'fast' with the number of message exchange rounds.

The third property – distribution – is important because it eludes the need for detecting faulty coordinators, leaders or token-holders. This detection usually has a price in terms of time and messages transmitted. Moreover, even a common failure like a process crash cannot be detected in a strictly asynchronous system, since there are no bounds on the communication delays.

The *resilience* of a protocol can be defined as the maximum number of faults in the presence of which the protocol still behaves according to its specification. The optimal resilience for asynchronous consensus has been shown to be $\lfloor \frac{n-1}{3} \rfloor$ [5]. Atomic broadcast is an equivalent problem, so the optimal resilience is the same [19]. Optimal resilience is an important property because the need for additional processes to tolerate the same number of faults involves a cost in terms of additional resources (e.g., additional hardware).

The evaluation of a distributed protocol is usually made in terms of time and message complexities. Time complexity is usually considered more important, so we evaluate the protocols in terms of this criterion. In asynchronous systems, time complexity is usually measured in terms of maximum number of *asynchronous rounds*. An asynchronous round involves a process sending a message and receiving one or more messages sent in response to the former. For randomized protocols, the metric is usually the *expected number of asynchronous*

rounds. Our multi-valued consensus protocol has time complexity $O(1)$, i.e., it has a constant expected number of rounds. The complexities of the vector consensus and the atomic broadcast protocols are both $O(f)$. These complexities are at least as good as previous works, except for one vector consensus that manages to have time complexity $O(1)$ at the cost of a higher message complexity [3].

Paper organization. The paper is organized as follows. The following section defines the system model and the two components used by our protocols: reliable broadcast and binary consensus. Section 3 presents our multi-valued consensus protocol and proves its correctness. Sections 4 and 5 present, respectively, the vector consensus and atomic broadcast protocols. Section 6 assesses the performance of the protocols. Section 7 discusses some related work and Section 8 concludes the paper.

2 Definitions

2.1 System Model

The system is composed by a set of n processes $P = \{p_1, p_2, ...p_n\}$. A process is said to be *correct* if it does not *fail* during the execution of the protocol, i.e., if it follows the protocol. We assume that at most $f = \lfloor \frac{n-1}{3} \rfloor$ processes can fail and we call these processes *corrupt*. These failures can be Byzantine, meaning that processes can stop, omit messages, send incorrect messages, send several messages with the same identifier, etc. Additionally, corrupt processes can pursue their goal of breaking the properties of the protocol alone or in collusion with other corrupt processes.

Processes are fully-connected by reliable channels with two properties: if the sender and the recipient of a message are both correct then (1) the message is eventually received and (2) the message is not modified in the channel. In practice, these properties have to be obtained with retransmissions and using cryptography. The channels are assumed to be *fair*, i.e., if a message is sent infinitely often by a process, then it is received infinitely often by its recipient. Message authentication codes (MACs) are cryptographic checksums that serve our purpose, and only use symmetric cryptography [22][2].

The system is asynchronous, which means that there are no clocks, no bounds on the processing times and on the communication delays.

2.2 Reliable Broadcast

A reliable broadcast protocol ensures essentially that all correct processes deliver the same messages, and that messages broadcasted by correct processes are delivered. Moreover, it ensures that no two different messages with the same identifier are delivered. This identifier includes the typical information in a protocol header: protocol type, sender, broadcast channel, and sequence number. An example of an asynchronous Byzantine-resistant reliable broadcast protocol is the one proposed by Bracha and Toueg [5]. We consider that the reliable

[2]The processes have to share symmetric keys in order to use MACs. In the paper we assume these keys are distributed before the protocol is executed. In practice, this can be solved using key distribution protocols available in the literature. This issue is out of the scope of the present paper.

broadcast is executed by calling the function R_Broadcast(M) (see, e.g., Algorithm 1 below).

Formally, a reliable broadcast protocol can be defined in terms of the following properties [19, 6]:

- *RB1 Validity:* If a correct process broadcasts a message M, then some correct process eventually delivers M.
- *RB2 Agreement:* If a correct process delivers a message M, then all correct processes eventually deliver M.
- *RB3 Integrity:* For any identifier ID, every correct process p delivers at most one message M with identifier ID, and if *sender(M)* is correct then M was previously broadcast by *sender(M)*.

The predicate *sender(M)* gives the field of the message header that identifies its sender. Note that we consider that the sender also delivers the messages it broadcasts.

2.3 Binary Consensus

A binary consensus protocol performs consensus on a binary value $b \in \{0, 1\}$. The problem can be formally defined in terms of three properties:

- *BC1 Validity:* If all correct processes propose the same value *b*, then any correct process that decides, decides *b*.
- *BC2 Agreement:* No two correct processes decide differently.
- *BC3 Termination:* Every correct process eventually decides.

Besides satisfying this definition, the binary consensus protocol to be used in the hierarchy has to be compatible with the structural properties given in the introduction: it cannot use public-key signatures, has to be asynchronous, has to take decisions in a distributed way and has to have optimal resilience. Examples of protocols that satisfy these requirements are [4, 8].

We consider that the binary consensus protocol is executed by calling the function B_Consensus(b, $bcid$), where b is the binary value proposed and $bcid$ the protocol execution identifier.

3 Multi-Valued Consensus

The first protocol of the hierarchy proposed in the paper is a multi-valued consensus. The definition of the problem is similar to the binary consensus, except that processes can propose values with arbitrary length $v \in \mathcal{V}$ ($\mathcal{V}$ is the domain of values that can be proposed). The protocol can decide one of the values proposed or a default value $\perp \notin \mathcal{V}$, in case the correct processes did not propose the same value. The definition is:

- *MVC1 Validity 1.* If all correct processes propose the same value v, then any correct process that decides, decides v.

- *MVC2 Validity 2.* If a correct process decides v, then v was proposed by some process or $v = \perp$.
- *MVC3 Validity 3.* If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.
- *MVC4 Agreement.* No two correct processes decide differently.
- *MVC5 Termination.* Every correct process eventually decides.

The problem of multi-valued consensus is usually stated in terms of the properties MVC1, MVC2, MVC4 and MVC5 above. We strengthen this definition with property MVC3 that states that the protocol does not decide values proposed only by corrupt processes. In practice, this property is satisfied by not deciding on a value if only f or less processes proposed it. This property is a requirement of the vector consensus protocol, implemented on top of the multi-valued consensus. However, the reader should notice that some consensus protocols in the literature have this property, even if they do not state it explicitly. For instance, binary consensus protocols that guarantee MVC1 also satisfy MVC3: if v is proposed only by corrupt processes, then all correct processes proposed *not v*.

3.1 The Protocol

The protocol is presented in Algorithm 1. Local variables are designated by lowercase letters with a subscript indicating the process to which they belong: w_i, b_i, c_i in process p_i. Vectors have one entry per process in P and are designated by an uppercase letter, e.g., vector V_i has entries $V_i[1]$, $V_i[2]$,... $V_i[n]$. The function $\#_x(V)$ counts the number of occurrences of x in vector V. The maximum number of faulty processes is a function of the total number of processes n: $f = \lfloor \frac{n-1}{3} \rfloor$. The protocol uses two types of messages: INIT and VECT. The content of messages is represented inside angles: $\langle ... \rangle$. A set called INIT_delivered$_i$ is used to store the received INIT messages. A call to *return* causes the termination of all the protocol's tasks. The value returned is the result of the protocol, i.e., the value decided.

Function `M_V_Consensus` is called with two arguments: the value proposed by the process (v_i) and the consensus identifier (*cid*). There is an initialization and tasks T1 and T2 are started concurrently (lines 1-2). Task T1 does most of the work, while task T2 simply receives INIT messages and stores them in INIT_delivered$_i$ (lines 22-23).

Task T1 begins by reliably broadcasting an INIT message with the value v_i proposed by process p_i (line 3). The identifier of the message includes the message type (INIT), the consensus identifier (*cid*) and the sender (*i*). Then, the task waits for the reception of $(n - f)$ INIT messages (including its own) and stores the proposed values in vector V_i (lines 4-5). The reliable broadcast protocol guarantees that two correct processes p_i and p_j do not receive different proposals from the same process (see Section 2.2). However, V_i can be different from V_j since the first $(n - f)$ INIT messages received by the two processes do not have to be the same.

If all correct processes propose the same value v then all correct processes receive at least $(n - 2f)$ INIT messages with v. If a process receives this number of messages with a value v, then it selects this value (lines 6-7) and reliably broadcasts it to all processes together with the vector V_i that justifies the selection (line 10).

Algorithm 1 Multi-valued Consensus protocol (for process p_i).

Function `M_V_Consensus` (v_i, cid)

INITIALIZATION:

1: INIT_delivered$_i \leftarrow \emptyset$; {INIT messages delivered}
2: **activate task** (T1,T2);

TASK T1:

3: `R_Broadcast` ($\langle$INIT, v_i, cid, i$\rangle$);
4: **wait until** (at least $(n-f)$ INIT messages have been delivered);
5: $\forall_j$: **if** ($\langle$INIT, v_j, cid, j$\rangle$ has been delivered) **then** V_i[j] $\leftarrow v_j$; **else** V_i[j] $\leftarrow \perp$;
6: **if** ($\exists_v : \#_v(V_i) \geq (n-2f)$) **then**
7: $w_i \leftarrow$ v;
8: **else**
9: $w_i \leftarrow \perp$;
10: `R_Broadcast` ($\langle$VECT, w_i, V_i, cid, i$\rangle$);
11: **wait until** (at least $(n-f)$ *valid* messages $\langle$VECT, w_j, V_j, cid, j$\rangle$ have been delivered);
12: $\forall_j$: **if** ($\langle$VECT, w_j, V_j, cid, j$\rangle$ has been delivered) **then** W_i[j] $\leftarrow w_j$; **else** W_i[j] $\leftarrow \perp$;
13: **if** ($\forall_{j,k}$ W_i[j] $\neq$ W_i[k] $\Rightarrow$ W_i[j] = $\perp$ or W_i[k] = $\perp$) and ($\exists_w$: $\#_w(W_i) \geq (n-2f)$) **then**
14: $b_i \leftarrow 1$;
15: **else**
16: $b_i \leftarrow 0$;
17: $c_i \leftarrow$ `B_Consensus`(b_i, cid);
18: **if** (c_i = 0) **then**
19: **return** $\perp$;
20: **wait until** (at least $(n-2f)$ *valid* messages $\langle$VECT, v_j, V_j, cid, j$\rangle$ with v_j = v have been delivered);
21: **return** v;

TASK T2:

22: **when** m_i = $\langle$INIT, v_j, cid, j$\rangle$ is delivered **do**
23: INIT_delivered$_i \leftarrow$ INIT_delivered$_i \bigcup \{m_i\}$;

Otherwise, it selects the default value $\perp$, which it also broadcasts. After broadcasting this message (VECT), the process waits for $(n - f)$ *valid* VECT messages, i.e., messages known to have a vector with real proposals and a value substantiated by those proposals. The identifier of a message VECT includes the protocol type (VECT), the the consensus identifier (*cid*) and the sender (*i*).

Definition 1 *A message $\langle$VECT, w_j, V_j, cid, j$\rangle$ is said to be* valid *iff:*

- $\forall_k$, $V_j[k] = \perp$ *or there is a message* $\langle$*INIT,* v_k, *cid, k*$\rangle \in$ *INIT_delivered so that* $V_j[k] = v_k$
- $w_j \neq \perp \Leftrightarrow \exists_w$ *:* $(\#_w(V_j) \geq (n - 2f)$ *and* $w_j = w)$

If the process does not receive two VECT messages with different values $w \neq w'$, and it receives at least $(n - 2f)$ messages with w, it proposes 1 for the binary consensus, otherwise it proposes 0 (lines 13-16). If the binary consensus decides 0, the vector consensus protocol decides on the default value $\perp$ (lines 17-19). If the binary consensus decides 1, the process waits for $(n - 2f)$ valid VECT messages with the same value w (line 20), in case it had not received them yet, and delivers this value (line 21). Notice that line 20 does not wait for $(n - 2f)$ messages but rather until it received cumulatively from the beginning $(n - 2f)$ VECT messages with the same value w.

3.2 Correctness Proof

The protocol in Algorithm 1 is correct if it satisfies properties MVC1 to MVC5. A preliminary result is given by the following lemma:

Lemma 1 *If a message $\langle$VECT, w_i, V_i, cid, i$\rangle$ is reliably broadcasted by a correct process p_i, then eventually all correct processes will consider it* valid.

Proof: The INIT messages are reliably broadcast (line 3). Consequently, all correct processes eventually deliver the same INIT messages (properties RB1-RB3 in Section 2.2). A correct process only puts in V_i values v_j it received in INIT messages (line 5). Therefore, for every value in a VECT message sent by a correct process, there is a INIT message that is eventually delivered by all correct processes. Additionally, a correct process always sends VECT messages with at least $(n - f)$ values (lines 4-5, 10). This proves the lemma, attending to the definition of *valid* message. □

Theorem 1 *(Validity 1) If all correct processes propose the same value v, then any correct process that decides, decides v.*

Proof: If all correct processes propose the same value v, then all processes deliver at least $(n - 2f)$ INIT messages with v (at most f processes are corrupt). Consequently, all correct processes make $w_i = v$, and send this value in a VECT message (lines 6-10). Moreover, all correct processes deliver at least $(n - 2f)$ valid VECT messages in line 11 (Lemma 1). No valid VECT message can have $w_i \neq v$ since at most f (corrupt) processes send INIT messages with a value different from v. Therefore, all correct processes make $b_i = 1$ (lines 13-14). All correct processes start a binary consensus protocol (line 17) that decides 1 (property BC1). The value decided is necessarily v (lines 18-21). □

Theorem 2 *(Validity 2) If a correct process decides v, then v was proposed by some process or $v = \perp$.*

Proof: The proof is obtained with a trivial inspection of the protocol. □

Theorem 3 *(Validity 3) If a value v is proposed only by corrupt processes, then no correct process that decides, decides v.*

Proof: The proof is by contradiction. If a correct process decides v then it received at least $(n - 2f)$ valid VECT messages with v. For a VECT message to be valid there has to be at least $(n - 2f) > f$ INIT messages with v, but the theorem assumes only corrupt processes propose v: a contradiction. □

Theorem 4 *(Agreement) No two correct processes decide differently.*

Proof: All correct processes get the same decision from the binary consensus protocol. The proof can be divided in two cases, depending on the value c_i decided by the binary consensus (line 17):

c_i = 0: All correct processes decide $\perp$ (lines 18-19).

c_i = 1: The proof is by contradiction. Two correct processes p_1 and p_2 decide differently if: (1) p_1 delivers $(n - 2f)$ valid VECT messages with the same value v_1 (line 20); and (2) p_2 delivers also $(n - 2f)$ valid VECT messages but with a value $v_2 \neq v_1$. Definition 1 guarantees that the vectors V in the VECT messages delivered by p_1 have $\#_{v_1}(V) \geq (n - 2f)$. In the most unfavorable case, we have exactly $\#_{v_1}(V) = (n - 2f)$. Something similar happens to p_2. Each value in the vectors in valid messages must be justified by an INIT message (Definition 1). Therefore, p_1 must have received $(n - 2f)$ INIT messages with v_1 and f INIT messages with $\perp$. The equivalent must have happen to p_2. Summarizing, we need $(n - 2f)$ INIT messages with v_1, $(n - 2f)$ INIT messages with v_2 and f INIT messages with $\perp$. The INIT messages are reliably broadcasted therefore there are at most n different INIT messages, so we must have: $(n - 2f) + (n - 2f) + f \leq n$. Solving this equation we have $n \leq 3f$. However, the paper assumes $f = \lfloor \frac{n-1}{3} \rfloor$ that implies that $n > 3f$. A contradiction.

□

Theorem 5 *(Termination) Every correct process eventually decides.*

Proof: Correct processes decide when they execute lines 19 or 21. The places of the protocol in which we have to prove that the protocol makes progress are the two executions of the reliable broadcast protocol (lines 3-4 and 10-11), the execution of the binary consensus protocol (line 17) and the reception of VECT messages in line 20.

The termination of the reliable broadcast protocol is guaranteed by its Validity and Agreement properties (RB1, RB2). All correct processes eventually deliver $(n - f)$ INIT messages in line 4 because all correct processes reliably broadcast an INIT message in line 3, and there are at most f corrupt processes. This proves that the protocol makes progress in lines 3-4. The justification for lines 10-11 is identical. The binary consensus protocol executed in line 17 is guaranteed to terminate by property BC3.

The protocol waits for the condition in line 20 only if the binary consensus decides 1. If all correct processes had proposed 0 for the binary consensus, then the process would have decided 0 (lines 17-19). Therefore, at least one correct process proposed 1 for the binary consensus. A correct process proposes 1 for the binary consensus only if it delivered $(n - 2f)$ valid VECT messages with the same value w (second condition in line 13 and lines 11-12). The VECT messages are reliably broadcasted, therefore if a correct process delivers $(n-2f)$ valid VECT messages with w, then all correct processes eventually do the same. Therefore no correct process blocks in line 20 and all terminate. □

4 Vector Consensus

Vector consensus makes agreement on a vector with a subset of the proposed values, instead of a single value [13]. In systems where Byzantine faults can occur, the vector is useful only if a majority of its values were proposed by correct processes. Therefore, the decided vector needs to have at least $(2f + 1)$ values. This problem is ultimately an adaptation for asynchronous systems of the classical problem of interactive consistency defined for synchronous systems [25]. The difference between the two problems is that interactive consistency makes agreement on a vector with the values proposed by all correct processes, while vector consensus guarantees only that the majority of the values were proposed by correct processes. The reason for this difference is that in asynchronous systems it is not possible to ensure that the vector has the proposals of all correct processes, since they can be arbitrarily delayed.

Vector consensus can be defined in terms of the following properties:

- *VC1 Vector validity:* Every correct process that decides, decides on a vector v of size n:
  - $\forall_{p_i}$: if p_i is correct, then either *V[i]* is the value proposed by p_i or $\perp$;
  - at least $(f + 1)$ elements of V were proposed by correct processes.
- *VC2 Agreement:* No two correct processes decide differently.
- *VC3 Termination:* Every correct process eventually decides.

4.1 The Protocol

The protocol is implemented by the function `Vector_Consensus` presented in Algorithm 2. The arguments are the value proposed (v_i) and the vector consensus identifier (*vcid*). The protocol starts by reliably broadcasting a VC_INIT message with the value proposed by the process (line 2). This message is identified by the protocol type (VC_INIT), the vector consensus identifier (*vcid*) and the sender (*i*). Then, the protocol runs one or more rounds until a decision is made (lines 3-8).

The algorithm begins each round by waiting for the reception of $(2f + 1 + r_i)$ VC_INIT messages (line 4). Notice that line 4 does not restart from scratch waiting for the $(2f + 1 + r_i)$ messages, but rather waits until that number of messages has cumulatively been received since the beginning. Next, the process builds a vector W_i with the values it received from other processes (at least $(2f + 1)$ in round 0, $(2f + 2)$ in round 1, ...) and

proposes the vector for a multi-valued consensus (lines 5-6). The identifier of the multi-valued consensus is unique for each execution by using a combination of *vcid* and the round number, r_i.

VC_INIT is reliably broadcasted, therefore all correct processes will eventually receive the same VC_INIT messages and build identical W vectors. When enough processes propose the same W vector for the multi-valued consensus, W is decided by this protocol and immediately after by the vector consensus (lines 6-9).

Algorithm 2 Vector Consensus protocol (for process p_i).

Function `Vector_Consensus` (v_i, vcid)

1: $r_i \leftarrow 0$; {round number}
2: `R_Broadcast` ($\langle$VC_INIT, v_i, vcid, i$\rangle$);
3: **repeat**
4: **wait until** (at least $(2f + 1 + r_i)$ VC_INIT messages have been delivered);
5: $\forall_j$: **if** ($\langle$VC_INIT, v_j, vcid, j$\rangle$ has been delivered) **then** W_i[j] $\leftarrow v_j$; **else** W_i[j] $\leftarrow \perp$;
6: $V_i \leftarrow$ `M_V_Consensus` (W_i, (vcid,r_i));
7: $r_i \leftarrow r_i + 1$;
8: **until** ($V_i \neq \perp$);
9: **return** V_i;

4.2 Correctness Proof

The protocol in Algorithm 2 is correct if it satisfies the properties VC1, VC2 and VC3.

Theorem 6 *(Vector validity) Every correct process that decides, decides on a vector v of size* n*: (1) $\forall_{p_i}$: if p_i is correct, then either* V[i] *is the value proposed by p_i or $\perp$; and (2) at least $(f + 1)$ elements of V were proposed by correct processes.*

Proof: The values proposed by each process are reliably broadcasted so all correct processes eventually deliver the same values (lines 2 and 4). Any correct process calls `M_V_Consensus` in line 6 with a vector W_i that satisfies the two conditions of the theorem: (1) each entry j of the vector contains either the value proposed by process p_j or $\perp$; and (2) W_i has at least $(2f + 1)$ elements from which at least $(f + 1)$ were proposed by correct processes (at most f processes are corrupt). The value decided by the protocol (line 9) is the value decided on the last execution of the multi-valued consensus (line 6). This value is one of the values proposed (property MVC2) and cannot have been proposed only by corrupt processes (property MVC3). Therefore, the value must have been proposed by at least one correct process so the two conditions of the theorem are satisfied. □

Theorem 7 *(Agreement) No two correct processes decide differently.*

Proof: The value decided is equal to the value decided on the last execution of the multi-valued consensus (lines 5-6) and all correct processes execute the same sequence of multi-vector consensuses. Therefore, the theorem is a trivial consequence of the Agreement property MVC4 of the multi-valued consensus. □

Theorem 8 *(Termination) Every correct process eventually decides.*

Proof: Let p_i be any correct process. All VC_INIT messages reliably broadcasted by correct processes are eventually delivered by all correct processes (properties RB1-RB3). Process p_i executes one or more calls to `M_V_Consensus`, and each of these calls eventually terminates (property MVC5). Each round of the loop, p_i waits for another VC_INIT message (line 4) before engaging in the multi-valued consensus (line 6). If p_i does not leave the loop, the latest by round $r = n - (2f + 1)$ process p_i and all other correct processes propose for the multi-valued consensus a vector with the values from all processes. Therefore, in that round all correct processes propose the same vector, the multi-valued consensus decides a value different from $\perp$ (property MVC1) and the protocol terminates (lines 8-9). □

5 Atomic Broadcast

The problem of atomic broadcast, or total order reliable broadcast, is the problem of delivering the same messages in the same order to all processes. The definition of the problem is equal to the definition of reliable broadcast plus a total order property:

- *AB1 Validity:* If a correct process broadcasts a message M, then some correct process eventually delivers M.
- *AB2 Agreement:* If a correct process delivers a message M, then all correct processes eventually deliver M.
- *AB3 Integrity:* For any identifier ID, every correct process p delivers at most one message M with identifier ID, and if *sender(M)* is correct then M was previously broadcast by *sender(M)*.
- *AB4 Total order:* If two correct processes deliver two messages M_1 and M_2 then both processes deliver the two messages in the same order.

The identifier of an atomic broadcast message includes the protocol type (A_MSG), the message number (*num*) and the sender identifier (*i*).

The atomic broadcast protocol is implemented on top of the vector consensus protocol. It could also be implemented directly on top of the multi-valued consensus but, in the end, the functionality of the vector consensus protocol would have to be implemented in the protocol anyway. The approach we use is more modular and elegant, besides providing the two protocols, either of which may be useful for the system designer.

5.1 The Protocol

The protocol is presented in Algorithm 3. The initialization is carried out before the first transmission or reception of a message (lines 1-4). A process atomically broadcasts a message by calling the function `A_Broadcast`, which simply reliably broadcasts the message to all processes (lines 5-6). The message number *num* guarantees that all messages broadcasted by a correct process are unique, since this number is unique. If a malicious process tries to call `R_Broadcast` twice with the same message, then the reliable broadcast protocol delivers the message only once (see property RB3, Integrity).

The delivery of messages is handled by tasks T1 and T2. When a message is delivered by the reliable broadcast protocol, it is inserted in the set R_delivered$_i$ (lines 15-16). Whenever this set is not empty, the process tries to agree with the other processes on the delivery of the messages in the set (lines 7-14). The task starts by constructing a vector H$_i$ with a *hash* of each of the messages in R_delivered$_i$ (line 8). A hash works essentially as a fixed-length unique identifier of the message. The objective is to compress the input supplied to the vector consensus protocol, since the performance of this protocol depends on the size of the value (e.g., the communication time depends on the size of the messages). A hash is obtained using a *hash function* h defined by the following properties [22]:

- *HF1 Compression:* h maps an input x of arbitrary finite length, to an output $h(x)$ of fixed length.
- *HF2 One way:* for all pre-specified outputs, it is computationally infeasible to find an input that hashes to that output.
- *HF3 Weak collision resistance:* it is computationally infeasible to find any second input that has the same output as a specified input[3].
- *HF4 Strong collision resistance:* it is computationally infeasible to find two different inputs that hash to the same output.

The value proposed by a process to the vector consensus is itself a vector with the hashes of the messages, H$_i$ (lines 8-9). The vector consensus protocol decides on a vector X$_i$ with at least $(2f + 1)$ vectors H from different processes. If the hash of a message appears in at least $(f + 1)$ of these vectors, the process can be confident that the hash was proposed by at least one correct process (there are at most f corrupt processes), therefore there is no doubt that the message was reliably broadcasted to all processes. This is important because a malicious process might provide a hash for which there was no message to deliver. The process waits until all messages that are to be delivered are put in R_delivered$_i$ (line 10), then it stores them in A_deliver$_i$ (line 10). Finally, the process delivers the messages in A_deliver$_i$ in a pre-established order, removes them from R_delivered$_i$, and increments the atomic broadcast identifier (lines 12-14).

5.2 Correctness Proof

The atomic broadcast protocol in Algorithm 3 is correct if it satisfies the properties AB1, AB2, AB3 and AB4.

Theorem 9 *(Validity) If a correct process broadcasts a message M, then some correct process eventually delivers M.*

Proof: A correct process broadcasts a message M by calling `A_Broadcast`(m). Then, the atomic broadcast protocol adds a header to the message and broadcasts it using the reliable broadcast protocol (line 5). The

[3]A guessing attack is expected to break the property HF3 in 2^m hashing operations, where m is the number of bits of the hash. A birthday attack can be expected to break property HF4 in $2^{m/2}$ hashing operations. In a practical setting, a hashing function with 128 bits like MD5, or 160 bits like SHA-1, can be considered secure enough for our protocol. Nevertheless, we consider HF2, HF3 and HF4 to be assumptions.

Algorithm 3 Atomic Broadcast protocol (for process p_i).

INITIALIZATION:

1: R_delivered$_i \leftarrow \emptyset$; {messages delivered by the reliable broadcast protocol}
2: aid$_i \leftarrow 0$; {atomic broadcast identifier}
3: num$_i \leftarrow 0$; {message number}
4: **activate task** (T1,T2);

WHEN **Function** `A_Broadcast` (m) is called DO

5: `R_Broadcast` ($\langle$A_MSG, num$_i$, m, i$\rangle$);
6: num$_i \leftarrow$ num$_i$ + 1;

TASK T1:

7: **when** (R_delivered$_i \neq \emptyset$) **do**
8: H$_i \leftarrow$ {hashes of the messages in R_delivered$_i$};
9: X$_i \leftarrow$ `Vector_Consensus` (H$_i$, aid$_i$);
10: **wait until** (all messages with hash in $f + 1$ or more cells in vector X$_i$ are in R_delivered$_i$);
11: A_deliver$_i \leftarrow$ {all messages with hash in $f + 1$ or more cells in vector X$_i$};
12: atomically deliver messages in A_deliver$_i$ in a deterministic order;
13: R_delivered$_i \leftarrow$ R_delivered$_i$ - A_deliver$_i$;
14: aid$_i \leftarrow$ aid$_i$ + 1;

TASK T2:

15: **when** $\langle$A_MSG, num, m, i$\rangle$ is delivered by the reliable broadcast protocol **do**
16: R_delivered$_i \leftarrow$ R_delivered$_i \bigcup \{\langle$A_MSG, num, m, i$\rangle\}$;

properties of this reliable broadcast protocol ensure that all correct processes eventually receive M (properties RB1-RB3). This guarantees that there is an execution of the lines 7-14 when all correct processes put the hash of M in H (line 8), unless these processes already delivered M in a previous execution of line 12. When all correct processes put the hash of M in H, the vector consensus decides on a vector that includes at least $f + 1$ entries with that hash (property VC1, Vector validity). Therefore, if the protocol does not block, all correct processes deliver M (lines 10-12).

The protocol might block only in lines 9 and 10. It does not block in line 9 because the vector consensus is guaranteed to terminate (property VC3, Termination). Line 10 waits until all messages that have to be delivered by the atomic broadcast protocol (those with $f + 1$ hashes in the vector) are in R_delivered. A message with $f + 1$ hashes in the vector must have been already delivered by the reliable broadcast protocol to at least one correct process. Therefore, this protocol will eventually deliver the message to all correct processes (properties RB1-RB3), so no correct process blocks in line 10. □

Theorem 10 *(Agreement) If a correct process delivers a message M, then all correct processes eventually deliver M.*

Proof: The theorem starts from the fact that one correct process, say p$_i$, delivers M. Therefore: (1) the vector consensus in line 9 decides on a vector with at least $f + 1$ hashes of M; and (2) the reliable broadcast protocol delivers M to p$_i$, therefore it delivers M to all correct processes (properties RB1-RB3). All correct processes

get the same results from the vector consensus so all eventually deliver M. □

Theorem 11 *(Integrity) For any message M, every correct process p delivers M at most once, and if* sender(M) *is correct then M was previously broadcast by* sender(M).

Proof: The proof of the first assertion is trivial from the inspection of the algorithm. The proof of the second assertion follows directly from the properties of the communication channels. □

Theorem 12 *(Total order) If two correct processes deliver two messages M_1 and M_2 then both processes deliver the two messages in the same order.*

Proof: Any correct process delivers messages only after an execution of `Vector_Consensus`. All correct processes execute the same instances of the vector consensus protocol, identified by aid $= 0, 1, 2, ...$ The messages which are delivered are all those with at least $f+1$ hashes in the vector returned by `Vector_Consensus` and the order of delivery is deterministic (line 12). Therefore, all processes deliver the same messages in the same order. □

6 Performance Evaluation

The time complexity of the *multi-valued consensus protocol* is twice the number of rounds executed by the reliable broadcast protocol L_{rb} (lines 3 and 10) plus the time complexity of the binary consensus protocol L_{bc} (line 17). The reliable broadcast protocol by Bracha and Toueg runs in 3 rounds [5]. The time complexity of the binary consensus protocol is measured in expected number of rounds, since the protocol is randomized, therefore probabilistic. The protocol by Canetti and Rabin has a constant expected time, $O(1)$ [8][4], therefore the time complexity of the multi-valued consensus protocol is:

$$L_{mvc} = 2L_{rb} + L_{bc} = O(1) \tag{1}$$

The protocol can be optimized by replacing the second reliable broadcast in line 10 by a (normal) broadcast or by the transmission of the VECT message individually to all processes. In this case, one correct process might receive $(n - 2f)$ messages with the value to be decided v, while another correct process would not. To circumvent this problem, all correct processes that receive $(n - 2f)$ messages with the value v (line 11) must resend these messages to all other processes. This optimization reduces the 3 rounds of the reliable broadcast protocol to 2 rounds.

The *vector consensus protocol* runs in the best case in one round, in the worst in $n - (2f + 1) + 1$ rounds (e.g., if $n = 4, f = 1$, the protocol terminates in one or two rounds). In the best case the loop in lines 3-8 will be executed only once so the time complexity will be the sum of those of the reliable broadcast (line 2) and the multi-valued consensus (line 6). If the protocol does not terminate in the end of the first round, it is reasonable to expect that all VC_INIT messages reliably broadcasted will be delivered during the first execution of `M_V_Consensus`, since this consensus involves several rounds of message exchange (two

[4]The binary consensus protocol by Bracha has also an expected number of rounds of $O(1)$ if $f = O(\sqrt{n})$, but 2^{n-f} otherwise.

Protocol	Time complexity	Best case
Multi-valued consensus	$O(1)$	2 reliable broadcasts, 1 binary consensus
Vector consensus	$O(f)$	1 reliable broadcast, 1 multi-valued consensus
Atomic broadcast	$O(f)$	less than 1 reliable broadcast, 1 vector consensus

Table 1: Number of rounds of the three protocols.

reliable broadcasts plus one binary consensus). This would make the protocol terminate in the second round. However, a collusion of malicious protocols could try to delay the protocol a maximum of f rounds. Therefore, the time complexity of the algorithm is $O(f)$.

The time complexity of the *atomic broadcast protocol* depends on the amount of messages being transmitted. If only occasional messages are sent, the time complexity is equivalent to one reliable broadcast (line 5) plus one vector consensus (line 9), therefore the expected number of rounds is $O(f)$. However, if messages go on arriving during an execution of the vector consensus protocol, in the next round task T1 will try to make agreement on several messages instead of only one. Therefore this protocol exhibits the virtuous characteristic that its number of rounds declines considerably if the rate of transmissions increases. Table 1 summarizes the results for all protocols.

7 Related Work

The FLP impossibility result implies that any consensus protocol in a strictly asynchronous environment has to be randomized. Most randomized consensus protocols presented in the literature are binary. An exception is the multi-valued crash-tolerant protocol in [15]. Turpin and Coan presented a transformation from binary to multi-valued consensus for synchronous systems [31]. Toueg presented a transformation for asynchronous systems [30]. The main difference of this transformation to Algorithm 1 is that Toueg uses signatures, therefore it does not require a reliable broadcast primitive but a weaker echo broadcast protocol. His protocol has optimal resilience and has time complexity $O(1)$, but needs asymmetric cryptography. Cachin et al. proposed a similar transformation, but the algorithm is based on voting the selection of the value proposed by each successive process [6]. The protocol has optimal resilience but uses signatures and has a time complexity of $O(nf^2)$, therefore it scales much worse than ours that has time complexity $O(1)$.

Interactive consistency was defined as the problem of agreeing on a vector with one value per correct process [25]. However, in asynchronous systems it is not possible to differentiate slow from crashed processes, and with a Byzantine fault model it might also be impossible to distinguish malicious from crashed processes. Therefore, for Byzantine asynchronous systems the vector consensus problem was defined [13]. Two vector consensus protocols based on failure detectors have been specified in the meantime [13, 1]. Recently, Ben-Or and El-Yaniv presented a randomized vector consensus protocol with optimal resilience, time complexity O(1) and no signatures [3]. However, the message complexity is considerably higher than ours, since the protocol runs n multi-valued consensus protocols in parallel, while ours runs at most $n - (2f + 1) + 1$ multi-valued consensuses.

For the crash fault model, some transformations from multi-valued consensus to atomic broadcast have been defined [19, 10, 18]. Cachin et al. defined a transformation from multi-valued consensus to atomic broadcast for Byzantine faults, but they use cryptographic signatures [6]. Doudou et al. presented a transformation closer to ours [12]. It does not use signatures but it has higher communication complexity since it gives the full messages to the consensus module, instead of just small hashes.

To the best of our knowledge, no transformation from vector consensus to atomic broadcast is available in the literature. A collection of randomized atomic broadcast protocols can be found in [23]. These protocols rely on signatures to guarantee the authenticity of the messages and do not have optimal resilience. Other Byzantine-resistant atomic broadcasts for asynchronous systems can be found in Rampart [28] that uses signatures and SecureRing [20] that uses a signed token. BFT [9] does not use signatures when there are no faults, therefore it is very efficient. Unlike ours, all these three protocols need a failure detector to put away corrupt processes. Apart from the added complexity, the design of Byzantine failure detectors that are complete is still an open research issue.

8 Conclusion

This paper proposes a hierarchy of intrusion-tolerant or Byzantine-resistant protocols. These protocols form a coherent family, sharing effective and efficient structural properties: signature freedom, full asynchrony, distribution and optimal resilience.

The hierarchy shows a series of protocol transformations: from binary consensus to multi-valued consensus; from multi-valued consensus to vector consensus; from vector consensus to atomic broadcast. The objective is to provide a modular set of protocols that a designer can use in practice in the construction of intrusion-tolerant systems, especially in systems with limited resources like embedded or ubiquitous environments. Therefore, the protocols evade a set of characteristics that might constitute a shortcoming in a real system: the use of public-key cryptography, a known performance bottleneck in intrusion-tolerant systems; time assumptions, often vulnerable to some attacks; the existence of leaders or other 'privileged' processes, whose failure might be costly to detect.

The multi-valued consensus protocol terminates in a constant expected number of rounds. However, due to the severe nature of malicious faults, vector consensus is more effective as a system building block for security-related applications. The time complexity of the vector consensus proposed is $O(f)$. The time complexity of the atomic broadcast protocol is also $O(f)$, although the number of rounds can be considerably lower if there are several messages being transmitted. These results look extremely promising. In fact, they are at least as good as previous works. We plan to run detailed evaluations of the protocol hierarchy and present the results in a future paper.

References

[1] R. Baldoni, J. Helary, M. Raynal, and L. Tanguy. Consensus in Byzantine asynchronous systems. In *Proceedings of the International Colloquium on Structural Information and Communication Complexity*,

pages 1–16, June 2000.

[2] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[3] M. Ben-Or and R. El-Yaniv. Optimally-resilient interactive consistency in constant time. *Distributed Computing*, 2003. To appear.

[4] G. Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 154–162, August 1984.

[5] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[6] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In J. Kilian, editor, *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer-Verlag, 2001.

[7] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132, July 2000.

[8] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 42–51, 1993.

[9] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.

[10] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[11] M. Correia, N. F. Neves, L. C. Lung, and P. Veríssimo. Low complexity Byzantine-resilient consensus. DI/FCUL TR 03–25, Department of Informatics, University of Lisbon, August 2003.

[12] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: From crash-stop to Byzantine failures. In *International Conference on Reliable Software Technologies*, pages 24–50, May 2002.

[13] A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes. Technical Report 97/30, EPFL, 1997.

[14] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[15] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal. Randomized multivalued consensus. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Computing*, pages 195–200, May 2001.

[16] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, December 1995.

[17] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[18] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

[19] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Department of Computer Science, May 1994.

[20] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, November 2001.

[21] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[22] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[23] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150:75–111, 1999.

[24] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 152–162, July 2001.

[25] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[26] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of the Fourth European Dependable Computing Conference*, pages 44–61, October 2002.

[27] M. O. Rabin. Randomized Byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[28] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

[29] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[30] S. Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 163–178, August 1984.

[31] R. Turpin and B. A. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, February 1984.

[32] P. Veríssimo and C. Almeida. Quasi-synchronism: a step away from the traditional fault-tolerant real-time system models. *Bullettin of the Technical Committee on Operating Systems and Application Environments*, 7(4):35–39, 1995.

[33] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.